How to predict Customer Churn using Autoencoders in Python

Written by:- Antoniea Hylton, Niharika Bangur

04/28/2023

This paper talks about predicting customer churn using autoencoders. This paper will comprise of what an autoencoder is, three important types of autoencoders, and finally will go over an example.

What is an Autoencoder?

An autoencoder is an unsupervised learning technique that takes the help of using a Neural Network for the task of representation learning. A Neural Network is a computational model inspired by the structure and functioning of the human brain. It consists of interconnected nodes, called neurons, organized in layers. Each neuron receives input, applies a mathematical function to it, and produces an output. Through a process called training, neural networks can learn from data and adjust the strengths of connections between neurons to make predictions or solve complex tasks, such as image recognition or natural language processing.

There are two important functions that an autoencoder learns. An **encoder** that transforms the input data and a **decoder** that recreates the output data. The encoder compresses the data into a smaller representation and then the decoder reconstructs the original data from the compressed representation.

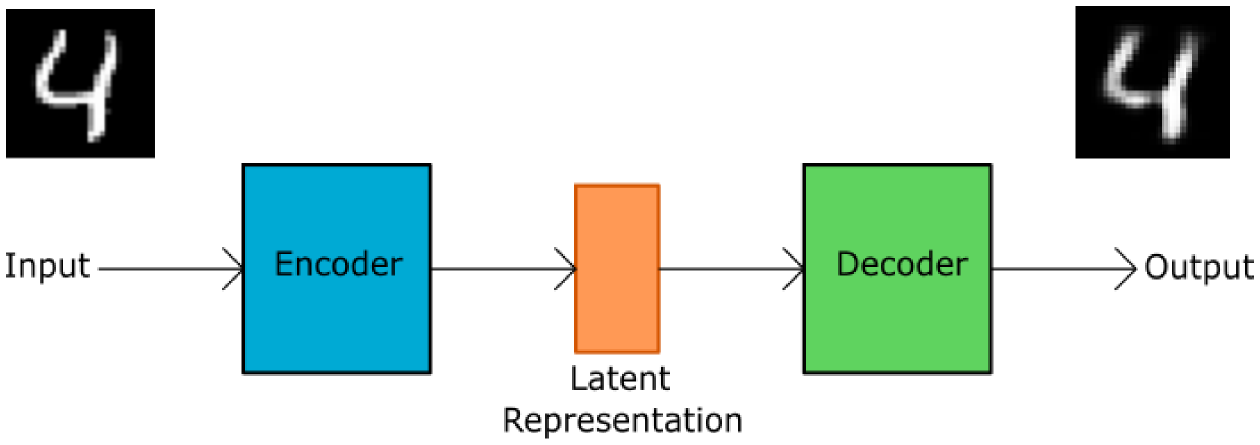


Image 1: example of encoder and decoder functions

During training the autoencoder tries to minimize the difference between the input data and the reconstructed data. For this, the network has to make sure to learn a compressed representation that makes sure to capture the most important information of the input data. Autoencoders can be used for tasks such as data compression, image denoising anomaly detection and feature extraction.

The main idea of an autoencoder is to recreate the input as best as possible. And this can be measured by the reconstruction loss.

An important section we have not talked about yet is the Hidden Layer called the Latent Space which is also known as **Bottleneck** and in *image 1* can be seen as Latent Representation. The reason for it being called a bottleneck is because it is a narrow part of the network where the data must pass through. This is the compression of the original data (input). The compressed data does have fewer dimensions than the original data, the most important features are kept, and others are discarded; this is what the Bottleneck layer is - where these features are preserved. The compressed representation is then used by the decoder to reconstruct the original data. *The main idea behind the bottleneck is to force the autoencoder to learn a compact and efficient representation of the input data.*

Three Important types of Autoencoders

1) Deep Autoencoders

Deep Autoencoders also known as Undercomplete Autoencoders consist of multiple hidden layers in the encoder and decoder networks. By adding these hidden layers, it allows the network to learn more complex and abstract representations of the input data. Each of these hidden layers in the **encoder** network takes the output from the previous layer and further compresses the input data. Each hidden layer in the **decoder** network takes the output from the previous layer and reconstructs the original data.

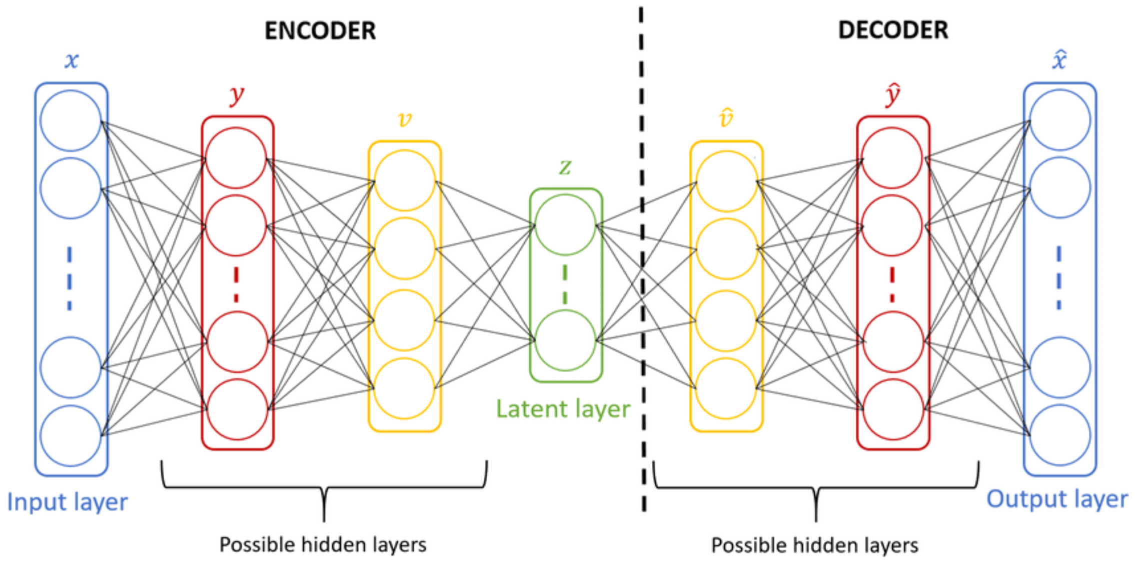


Image 2: Example of a deep autoencoder using a neural network

2) Sparse Autoencoders

Sparse autoencoders are a neural network that are designed to learn a compact and sparse representation of the input. They aim to identify the most important feature to reconstruct it which ignoring the less important features. This is to introduce sparsity constraints on the hidden layer which then encourages the network to learn a sparse representation of the input data. This can be achieved by techniques such as L1 Regularization. They can be more efficient in terms of storage and computation. They are also useful for feature extraction as the sparse representation can be used as input to other models. The latent layer is a compressed version of the original data set, so we can use our new data expecting that the compressed output contains the important features about the original observation.

3) Denoising Autoencoders

Denoising autoencoders, also called Noise Reduction, is the process of removing noise from a signal. They are designed to remove noise from the input data. They work by training the noise versions of the input data and then forcing them to reconstruct the original clean data. The denoising autoencoder introduces noise into the input data. The network is then trained to reconstruct the original data from the noisy input. For example, this helps a lot in making images clearer as seen in *image 3*.

Noisy Images   Denoised Images

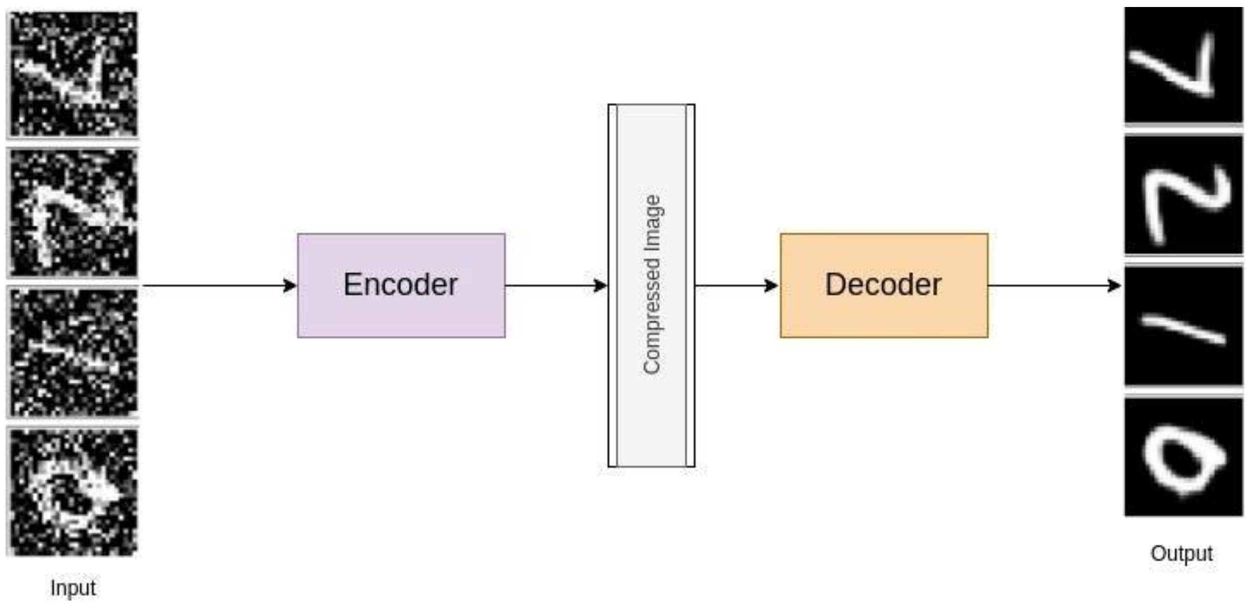


Image 3: Denoising Autoencoder

How to predict Customer Churn using Autoencoders

In today's highly competitive market we are able to see that customer churn has become a significant concern for businesses. Churn refers to the loss of customers or clients who have stopped doing business with a company. Detecting customer churn in advance is crucial for businesses to take proactive measures to retain their customers. This example is a Deep Autoencoder Model because this will be trained on a large dataset of Customers to extract the key features to detect the churned customers.

In this example we will use data about customers at a fictitious bank and use Python as our programming language.

Our analysis and predictions were executed in Google Colaboratory ("Colab") which is a product by Google where users can write and run Python code in a browser-based environment without the need for any local installation; however, this code can be ran in other environments that handle python as a coding language.

There are 7 steps to conduct this analysis:-

1) Reading the Data

In Colab, there are a few ways you can upload your data files. We clicked on the folder icon on the left-hand side, then dragged and dropped the file (uploading from your computer can also be done).

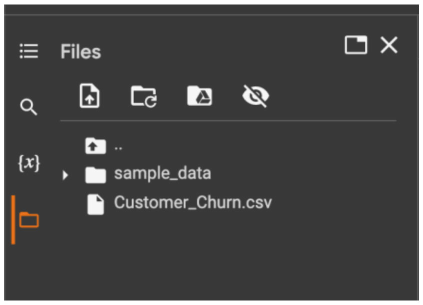
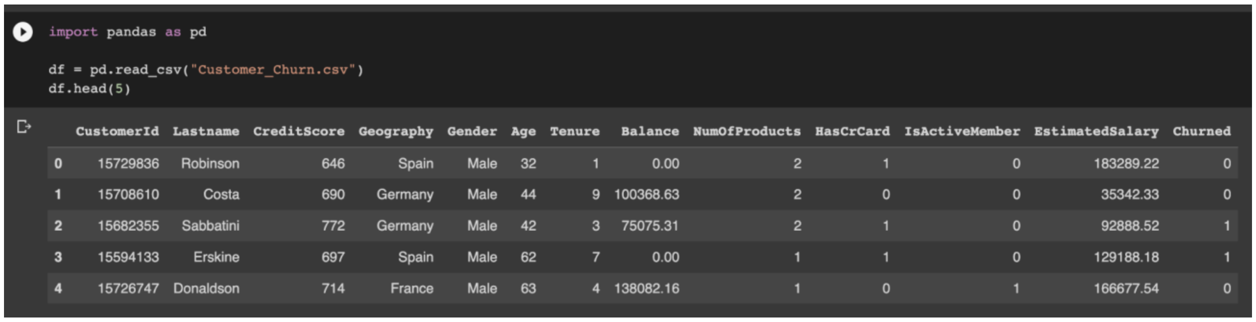


Image 4: Uploading

The first step is to read in the data and make sure that we understand the data and see what it consists of. The data used was called Customer_Churn.csv.

On the first line we are importing pandas which is a Python library which is part of a data manipulation dictionary. Next, we read in the dataset and then we take a closer look to show us the top 5 rows of a data set called df.

The "Churned" column is very important to us. In this column 1 = customer churned and 0 = customer did not churn. Now that we have seen what our data looks like we move onto the next step.



```
import pandas as pd

df = pd.read_csv("Customer_Churn.csv")
df.head(5)
```

	CustomerId	Lastname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Churned
0	15729836	Robinson	646	Spain	Male	32	1	0.00	2	1	0	183289.22	0
1	15708610	Costa	690	Germany	Male	44	9	100368.63	2	0	0	35342.33	0
2	15682355	Sabbatini	772	Germany	Male	42	3	75075.31	2	1	0	92888.52	1
3	15594133	Erskine	697	Spain	Male	62	7	0.00	1	1	0	129188.18	1
4	15726747	Donaldson	714	France	Male	63	4	138082.16	1	0	1	166677.54	0

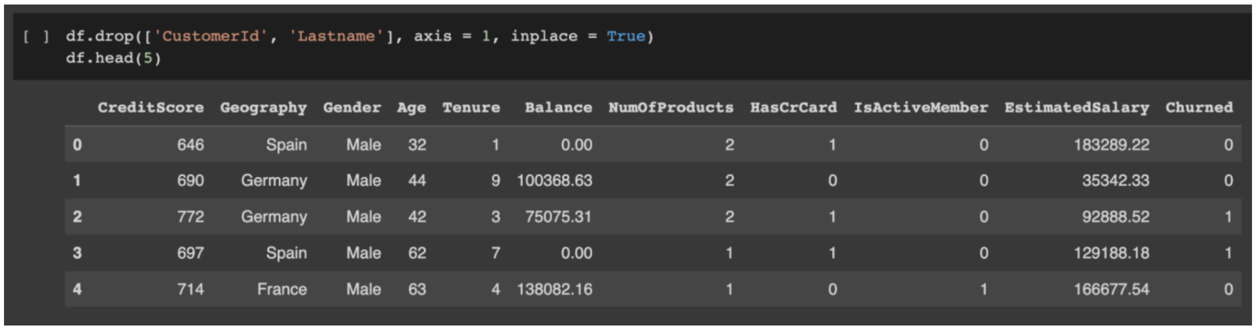
Image 5: Reading in the data

2) Dropping Unnecessary Variables

In this data there are some variables that are unnecessary to predict the number of customers churned. For us, “CustomerId” is not of much value and neither is the “Lastnames” column since they identify individuals, but do not provide any useful information in detecting customer churn, and hence they can be dropped.

We used the .drop() function and selected two columns. We used axis = 1 to indicate that we are removing columns and inplace=True to ensure that our dataframe (df) is modified in place, meaning the changes are applied directly to df.

As you can see in *image 6* the two columns are not present in the dataset anymore.



```
df.drop(['CustomerId', 'Lastname'], axis = 1, inplace = True)
df.head(5)
```

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Churned
0	646	Spain	Male	32	1	0.00	2	1	0	183289.22	0
1	690	Germany	Male	44	9	100368.63	2	0	0	35342.33	0
2	772	Germany	Male	42	3	75075.31	2	1	0	92888.52	1
3	697	Spain	Male	62	7	0.00	1	1	0	129188.18	1
4	714	France	Male	63	4	138082.16	1	0	1	166677.54	0

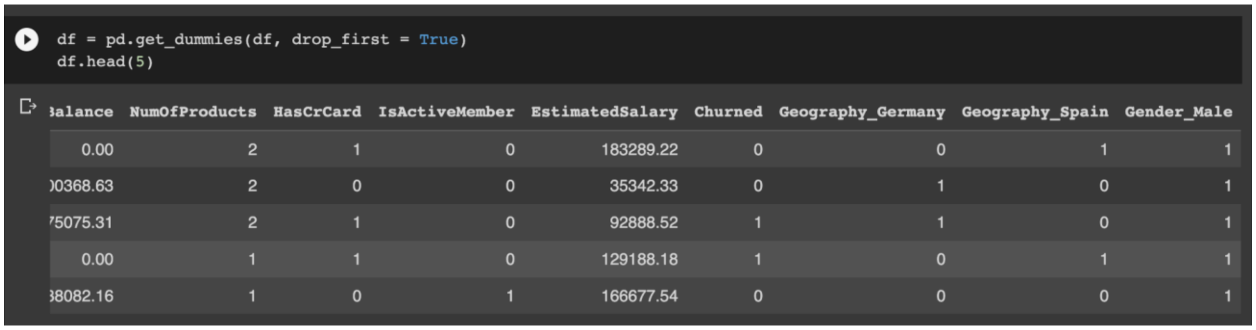
Image 6: Removing unnecessary variables

3) Dummy-encode categorical variables

Dummy-encoding is a technique used to represent the categorical variables as numerical in the data. In *image 7* the function .get_dummies doesn't need a column name to be mentioned since it makes dummies for all the categorical variables. The presence or absence will be indicated to let us know whether a specific category is present or is not present for each row in the dataset. We used the drop_first=True argument. It instructs the encoding process to drop the first level or category for each variable, resulting in n-1 dummy variables for a categorical variable with n distinct categories. By dropping the first level, we implicitly encode the presence or absence of each category relative to the first one. Extra columns are made for every unique category and the original columns are dropped.

In this dataset there are two categorical variables, “Geography” and “Gender”. We dummy-encoded both columns. “Geography” initially contained three countries: France, Germany, and Spain. After dummy encoding and dropping the first variable within the Geography column, we are left with a “Geography_Spain” and “Geography_Germany” column. If both values are 0, like in line 5 of *Image 7,* then that row corresponds to France and not Spain or Germany.

“Gender” initially contained Male or Female. After dummy encoding and dropping the first variable within the Gender column, we are left with Gender_Male;1 = male, and 0 = female.



```
df = pd.get_dummies(df, drop_first = True)
df.head(5)
```

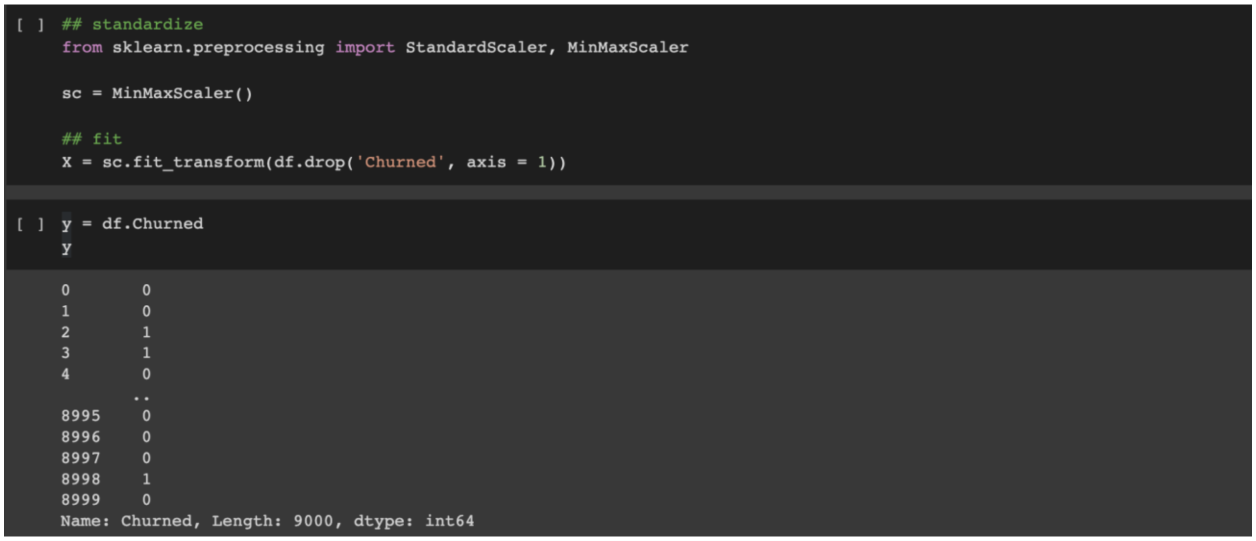
alance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Churned	Geography_Germany	Geography_Spain	Gender_Male
0.00	2	1	0	183289.22	0	0	1	1
0368.63	2	0	0	35342.33	0	1	0	1
5075.31	2	1	0	92888.52	1	1	0	1
0.00	1	1	0	129188.18	1	0	1	1
8082.16	1	0	1	166677.54	0	0	0	1

Image 7: Dummy-encoding

4) Scaling the Data

Scaling the data means transforming the numerical variables to a specific scale or range. This helps make sure that there is no bias and all the features contribute equally to the analysis and avoid any influence by outliers. The MinMaxScalar() function is a popular method of scaling the data in the range of (0,1). This happens by the maximum value minus the minimum value.

In *image 8* the fit_transform() method is used on the feature set X to transform the values of each feature to a range between 0 - 1 using the MinMaxScaler() function. The feature is set by dropping the variable “Churned”. This gets assigned to y to contain the value of the target variable.



```
[ ] ## standardize
    from sklearn.preprocessing import StandardScaler, MinMaxScaler

    sc = MinMaxScaler()

    ## fit
    X = sc.fit_transform(df.drop('Churned', axis = 1))
```

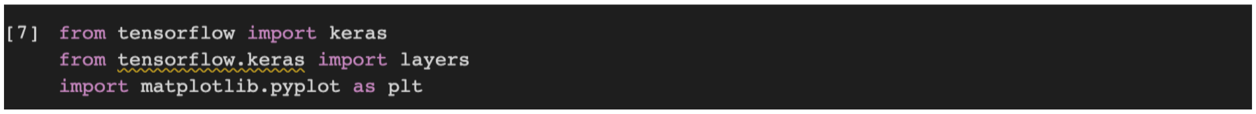
```
[ ] y = df.Churned
    y
```

```
0       0
1       0
2       1
3       1
4       0
       ..
8995    0
8996    0
8997    0
8998    1
8999    0
Name: Churned, Length: 9000, dtype: int64
```

Image 8: Scaling the data

5) Fitting an AutoEncoder Architecture

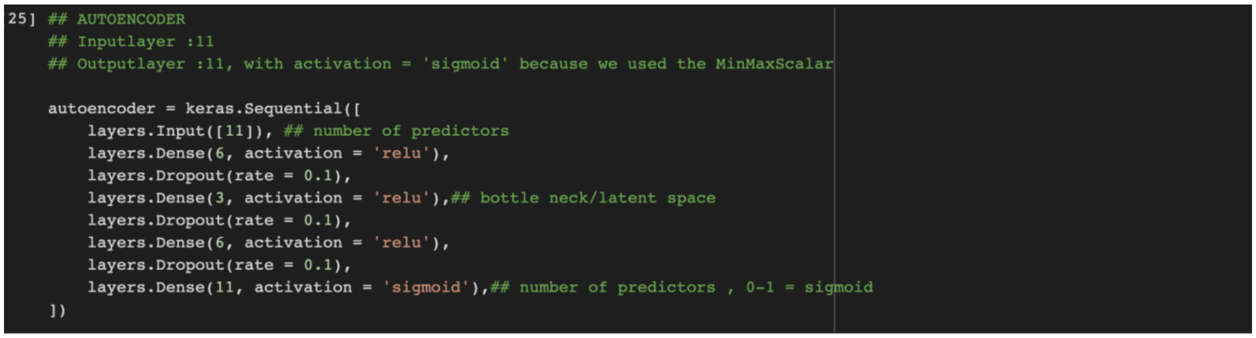
First we need to import some important libraries to be able to use certain functions.



```
[7] from tensorflow import keras
    from tensorflow.keras import layers
    import matplotlib.pyplot as plt
```

Image 9: Importing libraries

The *image 10* is the most important. Here we develop our deep autoencoder. Something to keep in mind is that “relu” and units = 0.1 helps in preventing overfitting in an autoencoder.



```
25] ## AUTOENCODER
    ## Inputlayer :11
    ## Outputlayer :11, with activation = 'sigmoid' because we used the MinMaxScalar

    autoencoder = keras.Sequential([
        layers.Input([11]), ## number of predictors
        layers.Dense(6, activation = 'relu'),
        layers.Dropout(rate = 0.1),
        layers.Dense(3, activation = 'relu'),## bottle neck/latent space
        layers.Dropout(rate = 0.1),
        layers.Dense(6, activation = 'relu'),
        layers.Dropout(rate = 0.1),
        layers.Dense(11, activation = 'sigmoid'),## number of predictors , 0-1 = sigmoid
    ])
```

Image 10: Autoencoder architecture

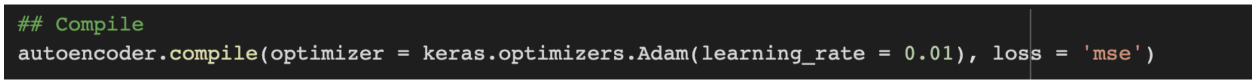
Table 1 provides a detailed description of what each line of the autoencoder architecture does.

	Layers	**Description**
Line 1	layers.Input([11])	Adds a layer (**input layer**) with 11 nodes which correspond with the number of predictors we have in our dataset.
Line 2	layers.Dense(6, activation = 'relu')	This adds a fully connected dense layer with 6 nodes
Line 3	layers.Dropout(rate = 0.1)	Randomly drops out about 10% of the input layer to prevent overfitting
Line 4	layers.Dense(3, activation = 'relu')	This is the bottleneck layer we talked about earlier in our blog. This adds a fully connected layer with 3 nodes.
Line 5	layers.Dropout(rate = 0.1)	Randomly drops out about 10% of to prevent overfitting
Line 6	layers.Dense(6, activation = 'relu')	This adds a fully connected dense layer with 6 nodes
Line 7	layers.Dropout(rate = 0.1)	Randomly drops out about 10% of to prevent overfitting
Line 8	layers.Dense(11, activation = sigmoid)	This adds our final layer (**output layer**) of 11 nodes. Here we use the sigmoid activation function to map out the values between 0,1 to prevent overfitting.

Table 1: Autoencoder architecture code description

We then compile the autoencoder by using the Adam optimizer and mean squared error (mse) as the loss. Compiling refers to configuring the autoencoder for training. When you compile an autoencoder, you specify how it should learn from the provided data.

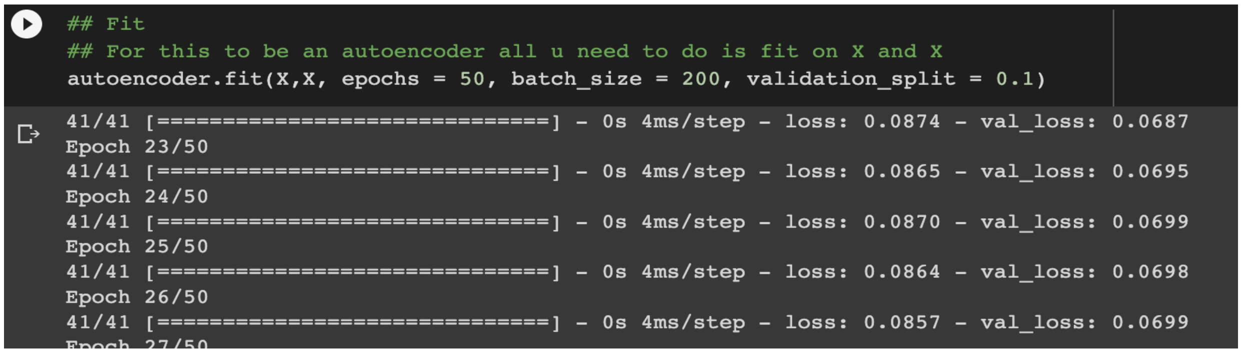
The reason for choosing mse was because it measures the average squared difference between the predicted values and the true values to measure how well the autoencoder model is able to reconstruct the input data.



```
## Compile
autoencoder.compile(optimizer = keras.optimizers.Adam(learning_rate = 0.01), loss = 'mse')
```

Image 11: Compiling the autoencoder

Next, moving onto fitting the model. The usage of X, X together is because the input data is used both as the input and the target output. Epochs is the number of times to iterate over the dataset. The batch_size specifies the number of samples that have been used in each batch to make sure of the memory usage. The validation split specifies the fraction of the training data will be set aside for validation.



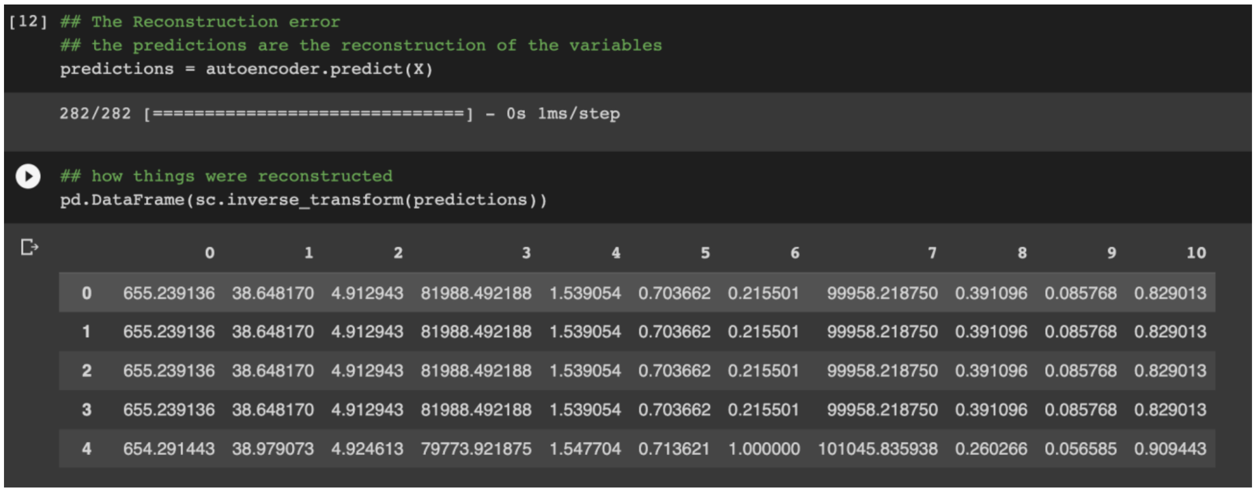
```
## Fit
## For this to be an autoencoder all u need to do is fit on X and X
autoencoder.fit(X,X, epochs = 50, batch_size = 200, validation_split = 0.1)
```

```
41/41 [==============================] - 0s 4ms/step - loss: 0.0874 - val_loss: 0.0687
Epoch 23/50
41/41 [==============================] - 0s 4ms/step - loss: 0.0865 - val_loss: 0.0695
Epoch 24/50
41/41 [==============================] - 0s 4ms/step - loss: 0.0870 - val_loss: 0.0699
Epoch 25/50
41/41 [==============================] - 0s 4ms/step - loss: 0.0864 - val_loss: 0.0698
Epoch 26/50
41/41 [==============================] - 0s 4ms/step - loss: 0.0857 - val_loss: 0.0699
Epoch 27/50
```

Image 12: Fitting the autoencoder

6) Determining the 500 most likely to churn customers

After fitting the autoencoder, we can use it to predict the customers who are most likely to churn. We can do this by selecting the customers whose reconstruction error is high. The reconstruction error is the difference between the original input data and the reconstructed data obtained from the autoencoder. We can sort the customers based on their reconstruction error and select the 500 observations with the highest error as the most likely to churn.



```
[12] ## The Reconstruction error
     ## the predictions are the reconstruction of the variables
     predictions = autoencoder.predict(X)
```

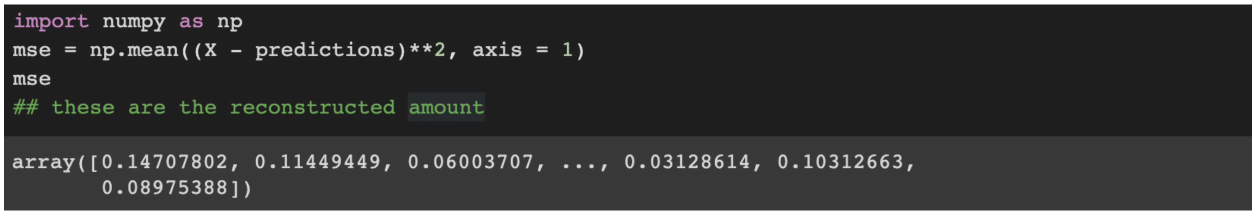
```
282/282 [==============================] - 0s 1ms/step
```

```
## how things were reconstructed
pd.DataFrame(sc.inverse_transform(predictions))
```

	0	1	2	3	4	5	6	7	8	9	10
0	655.239136	38.648170	4.912943	81988.492188	1.539054	0.703662	0.215501	99958.218750	0.391096	0.085768	0.829013
1	655.239136	38.648170	4.912943	81988.492188	1.539054	0.703662	0.215501	99958.218750	0.391096	0.085768	0.829013
2	655.239136	38.648170	4.912943	81988.492188	1.539054	0.703662	0.215501	99958.218750	0.391096	0.085768	0.829013
3	655.239136	38.648170	4.912943	81988.492188	1.539054	0.703662	0.215501	99958.218750	0.391096	0.085768	0.829013
4	654.291443	38.979073	4.924613	79773.921875	1.547704	0.713621	1.000000	101045.835938	0.260266	0.056585	0.909443

Image 13: Prediction for all customers

Now we can evaluate the reconstruction error for the input data “X”. The **2, in the code squares the difference between the input and the prediction for each level. The np.mean() function computes the average error across all samples.



```
import numpy as np
mse = np.mean((X - predictions)**2, axis = 1)
mse
## these are the reconstructed amount
```

```
array([0.14707802, 0.11449449, 0.06003707, ..., 0.03128614, 0.10312663,
       0.08975388])
```

Image 14: Reconstruction error in an array format

In *image 15* the histogram plot of the MSE shows the reconstruction errors that we computed above.

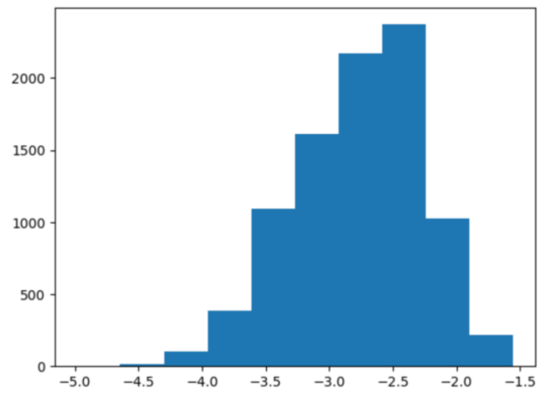
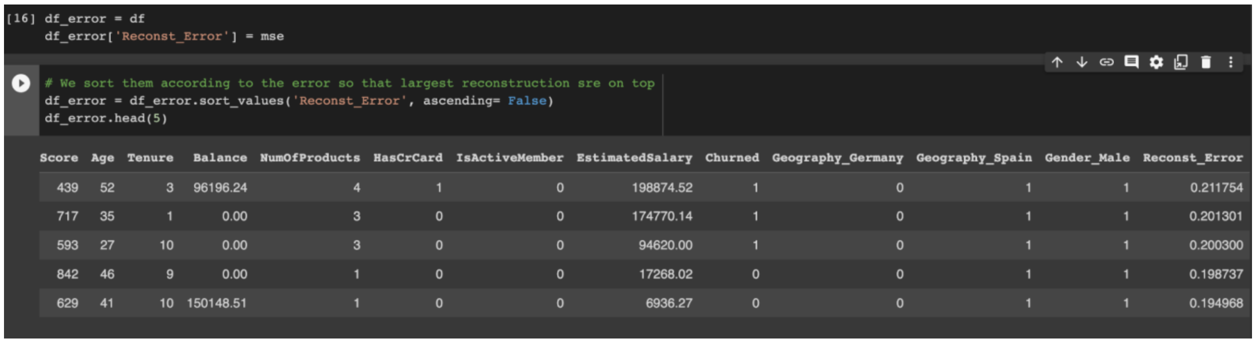


Image 15: Histogram of reconstruction errors

The reconstruction error values are then added as a new column into a new dataset called df_error. It also contains the columns from the prior data. The values are then sorted using the sort_values() function. See *Image 16.*



```
[16] df_error = df
     df_error['Reconst_Error'] = mse
```

```
# We sort them according to the error so that largest reconstruction sre on top
df_error = df_error.sort_values('Reconst_Error', ascending= False)
df_error.head(5)
```

Score	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Churned	Geography_Germany	Geography_Spain	Gender_Male	Reconst_Error
439	52	3	96196.24	4	1	0	198874.52	1	0	1	1	0.211754
717	35	1	0.00	3	0	0	174770.14	1	0	1	1	0.201301
593	27	10	0.00	3	0	0	94620.00	1	0	1	1	0.200300
842	46	9	0.00	1	0	0	17268.02	0	0	1	1	0.198737
629	41	10	150148.51	1	0	0	6936.27	0	0	1	1	0.194968

Image 16: Creating the new dataset df_error

We then extracted 500 observations with the highest reconstruction error values which would be those most likely to churn.

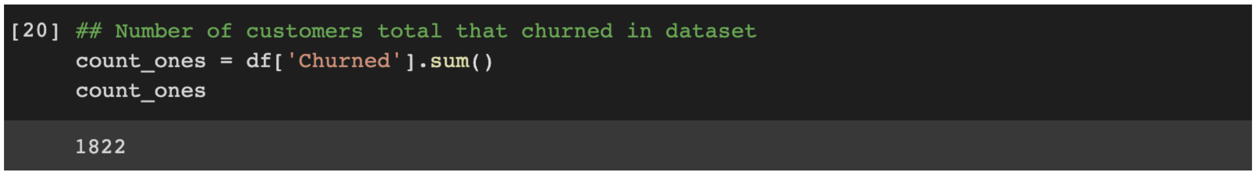
```
[18] top500 = df_error.head(500)

[19] top500.Churned.sum()

     185
```

Image 17: Number of customers churned from the top 500 Customers

With the result above we can see that 185 customers from the 500 observations with the largest errors churned.

7) Verifying the predictions

First in the whole dataset we need to see how many customers churned. This code counts the number of 1's in the "Churned" column because we know that 1 = churned.

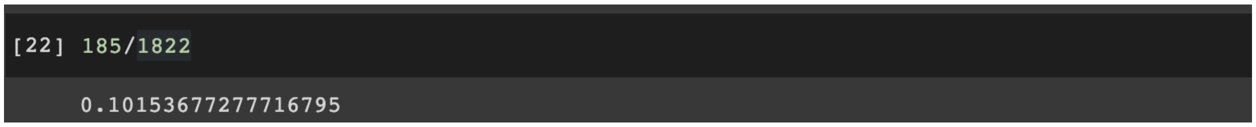


```
[20] ## Number of customers total that churned in dataset
     count_ones = df['Churned'].sum()
     count_ones

     1822
```

Image 18: Total number of customers churned in the dataset

In this last code we take the number of churned customers from the 500 observations with the largest errors and divide it by the number of customers that actually churned in the whole dataset which is 1822.

With this we can see that 10% of the customers were detected that churned out of the top 500 customers.



```
[22] 185/1822

     0.10153677277716795
```

Image 19: Customers detected

Final Thoughts

Through the help of an Autoencoder model we were able to detect the number of customers that churned. Autoencoders can be very helpful and can be used for many types of situations.

- *Data Compression:* Autoencoders can be used to compress the data and reduce its dimensionality. That can help companies store more data. This compression includes image and video compression as well.
- *Anomaly Detection:* Autoencoders can be trained on normal data and then used to detect anomalies or outliers in the new data. This helps in fraud detection.
- *Image generation:* Autoencoders can create new images that are similar to a set of training images. This is achieved by training an autoencoder on the training images and then using the decoder to generate the new image.
- *Denoising:* Autoencoders can help to remove noise from the data and then reconstruct the original data from the noisy input.
- *Feature Extraction:* Autoencoders can extract useful features from the data that can be used for other tasks.

CITATIONS FOR IMAGES

Image 2: Example of a deep autoencoder using a neural network

Roche, Fanny & Hueber, Thomas & Limier, Samuel & Girin, Laurent. (2019). Autoencoders for music sound modeling: a comparison of linear, shallow, deep, recurrent and variational models.

Image 3:Denoising Autoencoder

Nishad, Garima. “Reconstruct Corrupted Data Using Denoising Autoencoder (Python Code).” Medium, Analytics Vidhya, 25 Feb. 2021.